STUDY REPORT

This report is going to both include the documentation about the report and design decisions in a step by step manner as I have made some research along the way and built gradually since this was my first project using Spring Boot.

Project Base

First off, I have installed necessary tools for the development of a Spring project. Going on without any consideration of security, concurrency and performance issues at high load, I have created the endpoints in a day. At that time, the project is divided into packages:

> - entity
> - controller
> - repository
> - service
> - exception

There were several assumptions and design decisions made:

1. The functions generally return `id` rather than `name`.
2. The endpoints get the necessary info from `@RequestParam` (I created one endpoint using `@PathVariable` for demonstration).
3. Users are not able to **join** more than one team.
4. Likewise, the users are not able to **create** more than one team, as they are automatically joined a team during creation process.

Security

Over the weekend, I have researched in general how can I provide security and what are the steps and ways to do it. Most examples that I have found included password information and the security is gained by login. As our application does not have any password requiring login, I have decided to provide the security using JWT tokens.

Tokens

Upon creation, I have returned two tokens to the user in the response header. One is `access-token` and the other is `refresh-token`. The idea is that these tokens will be needed to use the APIs and a user will be able to refresh their access-token using their refresh-token (Note: refresh-token have much longer expiration date than access-token). While refreshing the access-token, we are also refreshing the refresh-token to prevent it from expiring.

To implement those mentioned above, I have generated a new package:

> - utility

There are two classes and one interface in that package. The interface named `TokenUtil.java` only contains the Algorithm used in the creation and verification of the token, and that interface is only reachable within the package.

Two classes are as follows:

- `TokenGenerator.java` : generates either an access or a refresh token and returns.
- `TokenValidator.java` : verifies if given `id` matches with the token subject.

After the implementation of that package, a new one is introduced:

> - config

This is needed to make configurations to the security. As mentioned in the Postman documentation, some endpoints are available to all and do not require any token. We are able to configure that certain request matchers are permitted to all.

```
http
.csrf(csrf -> csrf.disable())
.authorizeRequests(auth ->
  auth.requestMatchers("/user").permitAll();
  auth.requestMatchers(HttpMethod.GET).permitAll();
  })
```

With this configuration addition, the token mechanism was working well however, there were redundancy in the code as the generator and validator are called almost in every API call method. Then, I figured out that we are able to create a filter that filters every request coming to the program.

Filtering

Filters are middlewares that comes into play before every request *when they are added to the security filter chain*.

The `AuthenticationFilter.java` makes some necessary controls before the request is reaching to the actual methods. The main purpose of this class is to **validate** the token and disburden other methods from such action. This class is extending from `OncePerRequestFilter` and `@override doFilterInternal` method.

If not validated, it throws custom errors. However, they were not caught by the exception handler that I have written. I, then, realized that another filter is needed to resolve any exceptions that may occur in the security filter chain. This filter is named `FilterChainExceptionHandler.java` .

After creating those two filters, they are added to the filter chain by security config.

```
.addFilterBefore(new AuthenticationFilter(), BasicAuthenticationFilter.class)
.addFilterBefore(filterChainExceptionHandler, AuthenticationFilter.class)
```

Concurrency

Thanks to Spring Boot, adding concurrency to the methods was not so difficult. Methods of the `Service` classes are annotated with `@Async` and return types are changed to `CompletableFuture<T>`. Since the service methods are returning CompletableFuture objects, the `controller` methods are also adjusted in a way that they also return `CompletableFuture<T>`.

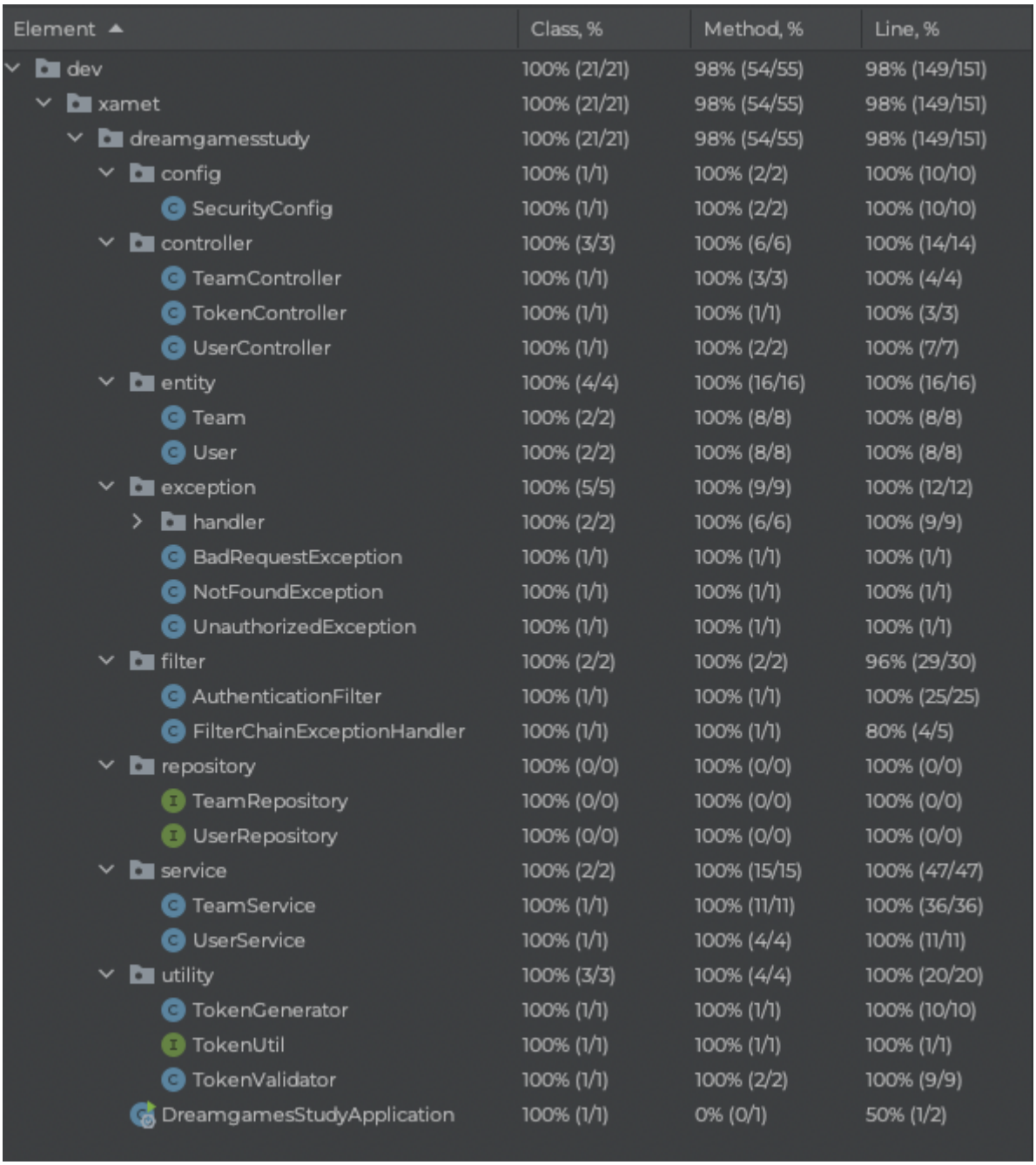
A simple sample code for `updateLevel` endpoint.

```
@PutMapping("/user/{id}")
public CompletableFuture<User> updateLevel(@PathVariable int id) {
    return userService.updateLevel(id);
}
```

```
@Async
public CompletableFuture<User> updateLevel(int id) {

    User user = getUserById(id);

    user.setLevel(user.getLevel() + 1);
    user.setCoins(user.getCoins() + 25);

    return CompletableFuture.completedFuture(userRepository.save(user));
}
```

Testing

I have spent some considerable amount of time to testing as I have no great prior experience in testing. While researching about unit testing in Spring, I have noticed that there is an option to test with `coverage`. As it shows, which methods and which lines are tested through its user interface, I have set myself a goal to test every possible method, and in the end I have managed to do it.



Element	Class, %	Method, %	Line, %
dev	100% (21/21)	98% (54/55)	98% (149/151)
xamet	100% (21/21)	98% (54/55)	98% (149/151)
dreamgamesstudy	100% (21/21)	98% (54/55)	98% (149/151)
config	100% (1/1)	100% (2/2)	100% (10/10)
SecurityConfig	100% (1/1)	100% (2/2)	100% (10/10)
controller	100% (3/3)	100% (6/6)	100% (14/14)
TeamController	100% (1/1)	100% (3/3)	100% (4/4)
TokenController	100% (1/1)	100% (1/1)	100% (3/3)
UserController	100% (1/1)	100% (2/2)	100% (7/7)
entity	100% (4/4)	100% (16/16)	100% (16/16)
Team	100% (2/2)	100% (8/8)	100% (8/8)
User	100% (2/2)	100% (8/8)	100% (8/8)
exception	100% (5/5)	100% (9/9)	100% (12/12)
handler	100% (2/2)	100% (6/6)	100% (9/9)
BadRequestException	100% (1/1)	100% (1/1)	100% (1/1)
NotFoundException	100% (1/1)	100% (1/1)	100% (1/1)
UnauthorizedException	100% (1/1)	100% (1/1)	100% (1/1)
filter	100% (2/2)	100% (2/2)	96% (29/30)
AuthenticationFilter	100% (1/1)	100% (1/1)	100% (25/25)
FilterChainExceptionHandler	100% (1/1)	100% (1/1)	80% (4/5)
repository	100% (0/0)	100% (0/0)	100% (0/0)
TeamRepository	100% (0/0)	100% (0/0)	100% (0/0)
UserRepository	100% (0/0)	100% (0/0)	100% (0/0)
service	100% (2/2)	100% (15/15)	100% (47/47)
TeamService	100% (1/1)	100% (11/11)	100% (36/36)
UserService	100% (1/1)	100% (4/4)	100% (11/11)
utility	100% (3/3)	100% (4/4)	100% (20/20)
TokenGenerator	100% (1/1)	100% (1/1)	100% (10/10)
TokenUtil	100% (1/1)	100% (1/1)	100% (1/1)
TokenValidator	100% (1/1)	100% (2/2)	100% (9/9)
DreamgamesStudyApplication	100% (1/1)	0% (0/1)	50% (1/2)

The most important thing that I have considered while creating tests that they should be in a way that mocks out external dependencies and focuses solely on the behaviour of itself. So, writing the tests, I have always tried to mock using mockito to improve the quality of the tests and increase the speed of each unit test by *not* depending other classes. For example, mocking `UserRepository` when testing `UserService` as I already know repository is working well and passing all the tests.

Conclusion

I have learned a lot within a week, and enjoyed what I did. Of course, I have faced some difficulties, for example, there were a static method on `TokenValidator.class`, and I didn't know that static methods need to be handled in a different way than those normal methods. That took some time to figure it out, however as I said it was fun in general.